REMARKS/ARGUMENTS

This amendment responds to the Office Action of February 7, 2007. Claims 1-45 are currently pending.

I. Objections

In claims 11, 14, 15, 26, 29, 30, 41, 44, and 45, the phrase "at least one of said plurality of objects" replaces the original language recited so as to provide proper antecedent basis. The dependencies of claims 32-45 have been changed to correspond to base claim 31.

II. Indefiniteness

The definiteness, under 35 U.S.C. 112, second paragraph, of claims 6-8, 12, 21-23, 27, 36-38, and 42 is questioned for use of the "trademark/trade name Java and/or Sun Microsystems." Citing Ex Parte Simpson, 218 U.S.P.Q. 1020 (Bd. App. 1982), the Action asserts that a claim using a trademark or trade name is uncertain because such a name does not identify any particular material or product but only a source of goods.

Applicant recognizes that Sun Microsystems, Inc. has registered the mark "Java." To dispel any possible confusion by the public, applicant has revised the specification so as to note the proprietary rights claimed by Sun in the Java name. Applicant has further amended claims 7, 22, and 37 to remove the term in connection with "native method interface" as such usage is redundant and unnecessary. Claims 8, 23, and 38, which depend from claims 7, 22, and 37, respectively, are likewise amended indirectly. In claims 6, 12, 21, 27, 36, and 42, which refer to "Java" or "Sun Microsystems," the trademark/trade name reference is both valid and appropriate in connection with "virtual machine" for reasons that will now be discussed.

As a preliminary note, it is not necessarily fatal to the definiteness of an application that a trademark or tradename is used. Rather, it is necessary that "each case" be "decided on its own facts." In re Metcalfe and Lowe, 161 U.S.P.Q.2nd 789, 792 (C.C.P.A. 1969). See also In re Gebauer-Fuelnegg et al., 50 USPQ 125 (C.C.P.A. 1941) where four claims referencing the trademark "pliolite" were held to be valid. In accord are Ex Parte Jerry Kitten (BPAI, heard Sept.

16, 1999, Appeal No. 19960513); and Ex Parte William Z. Goldstein (BPAI, heard June 7, 2005, Appeal No. 20050823). The last two "unpublished" decisions are viewable at http://des.uspto.gov/Foia/BPAIReadingRoom.jsp.

The term "Java virtual machine" had a recognized and particular meaning to those of ordinary skill in the art at the time the instant application was filed. According to the "Microsoft Computer Dictionary," Fourth Edition (Microsoft Press 1999), a Java Virtual Machine is an environment in which Java programs run. More technically, this term signifies a virtual machine that can interpret and execute Java bytecode, such as compiled from a Java application, and that can produce an output "customized" for a "particular platform," such as a host operating system. See Wikipedia, Java Virtual Machine, http://en.wikipedia.org/wiki/Java_virtual_machine. In short, the phrase "Java virtual machine" has an ordinary and customary meaning when examined "through the viewing glass" of a person skilled in the art. Home Diagnostics, Inc. v. Lifescan, Inc., 381 F.3d 1352, 1358, 72 USPQ2d 1276, 1279 (Fed. Cir. 2004). Thus the essential function of the second paragraph of §112 is fulfilled, which is to provide "adequate notice" to those of ordinary skill in the art of the "metes and bounds" of the invention. In re Goffe, 526 F2d. 1393, 1397, 188 USPQ 131, 135 (CCPA 1975) and In re Hammack, 427 F.2d 1378, 1382, 166 USPQ 204, 208 (CCPA 1970).

The complete specification for creating an official implementation of a Java Virtual Machine has been published in book format, see "Java™ Virtual Machine Specification," 2nd Edition, by Tim Lindholm and Frank Yellin (Prentice Hall 1999). Thus a "Kaffe" implementation of a "Java virtual machine" has been developed by a source working independently of Sun Microsystems in a "clean room" environment (e.g., working backward from only the specified functions of the "Java virtual machine"). See Wikipedia, Kaffe, http://en.wikipedia.org/wiki/Kaffe. There are further implementations from yet other sources. See Wikipedia, List of Java virtual machines, http://en.wikipedia.or/wiki/List_of_Java_virtual_machines.

A Java virtual machine developer may also seek certification by Sun Microsystems confirming that its particular implementation of a Java Virtual Machine is "compliant" (e.g., deserving of the steaming cup logo), which certification requires passing an appropriate technology compatibility test. Even in this situation, however, where the classification "Java virtual machine" is further refined and restricted so as to refer to a "compliant implementation" of such a machine, the term "Java" functions more as a certification mark (i.e., attesting to the product meeting certain performance standards) than as a regular trademark (i.e., indicating a unique source). It might be noted that the term "Java HotSpot" is how Sun refers to their own implementation of the Java virtual machine.

In claims 6, 21, and 36 the term "Java virtual machine" is not only definite but also necessary for distinguishing the scope of these claims over the broader term "virtual machine" as used in claims 1, 16, and 31. Although conceivably a coined or substitute term might be used to draw this distinction, such as "Java-enabled virtual machine," such a term does not have an established meaning to those of ordinary skill in the art. Reference to a "Java virtual machine" would be needed, in any event, to afford the substitute term a definite meaning. As there is no adequate substitute that carries the equivalent meaning, the term "Java virtual machine" is "as precise as the subject matter permits." See Shatterproof Glass Corp. v. Libbey-Owens Ford Co., 758 F.2d 613, 624, 225 USPQ 634, 641 (Fed. Cir. 1985).

Claims 12, 27, and 42 refer to a "Sun Microsystems virtual machine." The term "Java HotSpot" is how Sun refers to their own implementation of the Java virtual machine. One might object that the reference to Sun Microsystems places the "metes and bounds" of the subject matter of these claims under Sun's proprietary control where it can be bent and twisted like a "nose of wax." However, in the instant context, Sun is constrained by practical, if not legal, considerations from significantly altering (as opposed to extending) the basic features of its own implementation. If Sun were to make too large an alteration, its implementation would no longer be backward compatible with the legacy systems of its existing customers (thereby effectively breaching, for example, any license obligation to support its existing base of customers by providing current updates).

In the present context, then, use of the term "Java" or "Sun Microsystems" with the term "virtual machine" does not introduce indefiniteness in the claims. Further, as already noted, reference to "Java" in connection with "Native Method Interface" has been omitted in the amended claims. Hence, claims 6-8, 12, 21-23, 27, 36-38, and 42 meet the definiteness requirements of 35 U.S.C. 112, second paragraph.

III. Statutory Subject Matter

It is contended that claims 1-45 do not encompass statutory subject matter under 35 U.S.C. 101. It is asserted that these claims recite elements that can be "implemented in software alone" and are therefore "software, per se." It is also asserted that these claims "do not include hardware necessary to realize the functionality" of the software. Citation is made to MPEP §2106.01.

MPEP §2106.01 Sec. I. notes that "computer listings per se" are "not physical 'things.'" In other words, the reason why claims to "software per se" or "data structures per se" is found objectionable in the MPEP is because such claims are directed to "abstract ideas."

Another way of posing the "abstract idea" test is to ask whether the claimed invention can be performed purely as mental steps. Within the Patent Office itself, there has been some dispute as to whether this test is, in fact, appropriate (see Annex III c.(i), "The Mental Step Test," of the Interim Guidelines for Examination of Patent Applications for Patentable Subject Matter Eligibility, as published in the Official Gazette of November 22, 2005, available at www.uspto.gov/web/offices/com/sol/og/2005/week47/patgupa.htm). Be that as it may, the subject matter of applicant's claims possess sufficient "physicality," that is, they are not directed to mere abstract ideas or mental images. It will be noted, first, that each claim is directed to a "system" (not a process). This system is applied to computer applications designed for "concurrent operation" (the mind can hold only one thought at a time). Each of these applications operates in its own virtual "machine" (not mindspace). These applications share an object space selectively "connectable" to each computer application. Finally, as amended, the claims specify that the system is "computer-implemented" (e.g., by one or more physical computers). These limitations, separately or together, remove the subject matter of the claims from the realm of "abstract ideas" or mere "mental" images. Hence claims 1-45 recite statutory subject matter without transgressing on one of the judicially observed exceptions ("abstract ideas, natural phenomena, and laws of nature;" see MPEP §2106 Sec. IV.C.).

IV. Obviousness

As to claims 1-4, 6, 9-11, 13-19, 21, 24-26, 28-34, 36, 39-41, and 43-45, the Action asserts that Cranston et al. (U.S. 6,829,769 B2) teaches each claimed limitation (citing col. 3, lines 14-36) except for each computer application operating in its own virtual machine, but that such feature is shown by Galluscio et al. (U.S. 7,152,231 B1) and it would have been obvious to modify Cranston in view of Galluscio "because Cranston does not specify a programming language and Galluscio lists languages (col. 6, lines 5-17) that may be used in a similar communication system that uses a shared memory region and message queue [abstract]." As to claims 5, 20, and 35, the Action asserts that though Cranston fails to specifically teach a "last-in-first-out" queue, it would have been obvious to modify the combination of Cranston and Galluscio to supply such a queue, as taught by Martin et al. (7,017,160 B2), "because Cranston teaches using queues to manage object sharing [col. 4 lines 1-16] and Martin teaches other structures, such as a FIFO, can be used [col. 5 lines 39-58]." Finally, as to claims 7-8, 12, 22-23, 27, 37-38, and 42, the Action asserts that though Cranston fails to show a Java Native Method Interface, Jaworski teaches this feature, and it would have been obvious to modify the combination of Cranston and Galluscio to provide such an Interface "because Cranston combined with Galluscio teaches sharing objects with programs developed in different languages including C, C++ and Java and provides an example in C [Galluscio: col. 6 lines 5-30] and Jaworski teaches how to enable Java to use C and C++ programs for features not available in Java [page 984]."

IVa. Taken separately or together, Cranston and Galluscio fail to teach, as claimed, an object space "shared" by plural computer applications.

By way of general background, Cranston and Galluscio describe systems designed to facilitate interprocess communication. "Interprocess communication" generally refers to a distributed computing environment in which a first process passes messages, typically a request for service together with data, to a second process, which second process acts on the data in conformance with the request and returns the result to the first process. In programming languages of primarily procedural orientation, such as C or C++, interprocess communication over a network typically takes the form of a "Remote Procedure Call" (RPC) in which the first process passes a request, comprising the desired procedure and parameters, to the second process, which performs the procedure and returns the result. In certain object-oriented languages, such as Java, the equivalent procedure is a "Remote Method Invocation" (RMI) in which the first process passes a request, comprising a "method" invocation together with arguments, to the second process, which performs the method and returns the result. See, generally, Wikipedia, Inter-process communication, http://en.wikipedia.or/wiki/, Inter-process_communication.

To the extent, then, that one of ordinary skill might have had a reason to apply the teachings of Cranston or Galluscio, or both together, in a C++ or Java programming environment, they would have used these teachings to modify the message passing function (RPC or RMI) already in place between the distributed or separate applications (e.g., the "processes"). Each application, however, would have still operated within its own virtual machine, such as a conventional Java virtual machine as generally shown in FIG. 2. What would change, for example, in FIG. 2, is that one of ordinary skill, following Cranston and Galluscio, might have added additional functional boxes (e.g., a "shared memory" or "message heap" box together with one or more "queue" boxes) to the runtime memory area (indicated by the larger dashed region) of the one application and then permitted a second application (here conceptualized as a second FIG. 2) to access these additional functional boxes as necessary for providing the desired message passing function. In other respects, however, the virtual machine would have operated in the conventional manner and, in particular, the specific portion of the runtime memory area set aside by the one application as object space (heap 26) would have remained protected against access by any other application. Turning the object space of that one application into a *shared object* space, in the manner recited by subparagraph (a) of each independent claim, would have been regarded, under conventional thinking at the time of invention, as unjustifiably reckless because it would have been thought to expose that one application to any bugs, viruses, or other problems contained, inadvertently or maliciously, in the code of the outside application (e.g., see applicant's original disclosure, page 14, 2nd full paragraph). Such object space sharing would have been regarded as especially risky in a distributed environment of the sort exemplified by the Internet where the outside application or process may originate with an unknown host of uncertain trustworthiness. In short, even if one of ordinary skill had apparent reason to combine Cranston and Galluscio, the resulting combination would not have possessed, as claimed, an object space (as distinguished from a message space) "shared" by plural computer applications.

IVb. Taken separately or together, Cranston and Galluscio fail to show, as now claimed, a shared "object" space for holding a plurality of "updateable" objects.

Cranston and Galluscio both teach the use of "shared memory" for interprocess communication. In addition to this shared memory element (called a buffer in Galluscio), their technique relies on a queue (or, in Galluscio, a plurality of queues equal, in number, to the number of processes). The basic idea in both references is that a requesting process can submit a number of different requests to the shared memory, for example, all at once, without waiting for a response after each submission. At the same time it submits each request, the requesting process delivers a corresponding pointer to the queue (in Cranston, this is called "enqueing" a "process agnostic" memory handle, the implication being that any process, not just the specific process making the request, can now use this handle to access the corresponding "process specific" request; in Galluscio, the requesting process adds a "memory offset" to the queue of the particular servicing process it requires). This enables the servicing process to retrieve each request (e.g., by going to the shared memory location pointed to by the handle or offset) in the same ordered sequence as the queued pointers. For example, when a first-in-first-out queue is used, the servicing process can retrieve the requests in the order they were originally submitted and return the results in like order (with whatever interval is needed, between responses, for processing). Under this type of shared memory approach, the requesting process can continue to execute its sequence of operations without being forced to wait for a response each time it makes an interprocess service or procedure call (Cranston refers to such waiting as "blocking," see col. 2, lines 38-56). Also, less memory is needed than with certain other methods of interprocess communication. In certain other methods, for example, a copy of the entire request (the indicator for the requested service and input data) is moved from memory allocated to the requesting process to an intermediate memory area (e.g., a "pipe") and then to memory allocated to the servicing process so that, on each side of the transfer, the data is configured compatibly with the "virtual" or "dynamic" memory mapping used by the corresponding process (see col. 1, lines 24-31 of Galluscio).

Subparagraph (a) of each pending independent claim (claims 1, 16, and 31) requires a shared object space. While Cranston and Galluscio show a shared space for passing executable "instructions" (Cranston) or "messages" (Galluscio), neither reference shows a shared space for holding "objects," at least not in accordance with the normal scope given this term by those of skill in the art from at least the time of invention. Possibly, one might constrict the meaning of the term "objects" until it corresponds to "instructions" and "messages" insofar as each is reducible to some form of data, but this is like saying that the term "forests" corresponds to "woodpiles" as both are reducible to some form of "wood." It is would be more correct to say that each executable "instruction" or "message" of Cranston or Galluscio represents a single instruction similar to those included within a single executable thread because once such instruction or message is passed from the requesting process to the servicing process and used, by that servicing process, to compute a result, it has fulfilled its purpose. That is, after initiating its procedure calling or method invoking function, each instruction or message has, like a single-use matchstick, exhausted its purpose and is neither retained in nor returned to the shared memory in any form at all resembling that in which it left.

In applicant's disclosure, the closest thing to the "instructions" or "messages" of Cranston or Galluscio are represented by a single frame (not shown) of the Java Stack (see FIG. 2, item 28) insofar as that frame represents a method invocation by a single thread. In invoking a method, the thread provides the corresponding operational codes (opcodes) to the execution engine 18, which tell the engine what operations to perform, and passes any necessary arguments (operands) and local variables to the frame, which then provides the data on which the engine performs the requested operations. As one method calls another or as the thread calls multiple separate methods, the corresponding frames build up to form the Stack, and the execution engine executes the corresponding method or instructions relating to each frame in reverse calling sequence as retained by the Stack. This described operation, it may be noted, represents a conventional aspect of a Java Virtual Machine.

In somewhat similar fashion, each "instruction" (FIG. 7A, item 742) of Cranston is placed in the shared memory (or shared memory heap, SMH), which instruction comprises, an "operation code" (col. 8, lines 56-58) and "any data needed or useful in processing the instruction" (col. 8, lines 15-17; see also col. 7, lines 18-20). The shared memory queue, SMQ 214, of Cranston roughly performs the function of a conventional Java stack insofar as it determines the order of processing of the executable instructions. To extend the analogy still further, the "handler functions" of Cranston (as grouped together to form a "protocol" 230a or 230b) perform the processing or execution role. That is, whichever process (220a or 220b) is performing the servicing role for a particular instruction in Cranston hands off the task of executing the instruction to a selected one of the handler functions based on the service requested (as designated by the instruction's operational code, see col. 8, lines 54-56 and col. 9, lines 19-20). Even at a high level of abstraction, then, Cranston's system (with its queue) suggests nothing more than certain conventional elements (e.g., a Java stack) of a conventional Java virtual machine (see applicant's FIG. 2), which elements are not being claimed, as such, by applicant.

In contrast to such executable "instructions" or "messages," the term "objects," as conventionally used in software language technologies, normally refers to data structures that possess some degree of persistence or permanence, that is, they possess "attributes" or "states" in the same manner as the real world objects after which they are patterned. Thus the screen printout or display of a particular "window" object may be of varying height, width, screen position, and fill color. Though the user may perform predefined types of operations on this window object, for example, by resizing it and dragging it to a new position, the object continues to persist in recognizable form after the completion of each operation despite, for example, the modification of one or more of its attributes (such as screen position). Objects, in short, normally embody one or more persistent states (as represented by their "instance" variables) that are updateable in accordance with prespecified operations (as defined by their "class"). In order that the claims convey a comparable meaning where they refer to a shared object space for storage of a plurality of "objects," the term "objects" has been recast as "updateable objects" (support for this change may be found in applicant's original specification on page 15, lines 5-9 and on page 19, line 1). The executable "instructions" and "messages" of Cranston and Galluscio, respectively, are not "updateable objects" (once the instruction or message passes from the shared memory to the servicing process for execution, it is not restored to that memory in any equivalent form). Nor is the shared memory of Cranston or Galluscio, by itself, sufficient infrastructure to support such updateable objects (for example, Cranston and Galluscio do not disclose a "method area," as shown in FIG. 8 of applicant's drawings, for, inter alia, holding the method definitions that specify precisely the various ways in which the objects in shared memory space 200 may be updated; which method definitions are grouped for each updateable object under the aegis of a corresponding class definition, these class definitions being made available by being loaded into the virtual machine through a "Class Loader" subsystem, as also depicted).

IVc. There would have been no apparent reason, in the first instance, to combine Cranston and Galluscio.

It has been shown above that Cranston and Galluscio, either taken separately or together, fail to show each element of the pending claims; that is, they fail to show an object space "shared" by plural applications for holding a plurality of "updateable objects." Having said this, it should be noted that there would not, in the first instance, have been a reason apparent to those of ordinary skill for combining these two references as such combination would have rendered inoperable the intended purpose of each reference.

One of the declared features of Cranston is its "process agnostic" handles (col. 7, lines 15-16). Referring to FIG. 2 of Cranston, these process agnostic handles enable either process 220a or 220b to access an instruction submitted to the shared memory 212 by the other process 220b or 220a (e.g., by retrieving the handle corresponding to that instruction). Such handles, in other words, permit bi-directional operation and balanced distribution of computing tasks with a shared queue. This feature, however, brings certain tradeoffs. If any process can pull the handle for a stored instruction, then every process must possess the necessary resources to service any instruction. This, in turn, requires that each process provide a full complement of services (it will be noted, in FIG. 2, that the handler routines or service protocols, e.g., SMTP, IMAP4, DAV, POP3, and NNTP, provided by the process on one side match the routines or service protocols provided by the process on the other). This symmetry of operation, while, acceptable for some applications, is not readily adaptable to others including, for example, the more common client/server environment where the server has a much wider range of servicing capabilities than the client or in those distributed computing environments where high-resource hosts, such as mainframe servers, are linked with minimal resource hosts, such as palm pilots or smartcards.

The Galluscio system, on the other hand, is well-suited to the sort of nonsymmetrical distributed environment just described, that is, to those environments in which the various hosts possess widely differing servicing capabilities. Here again, however, there are tradeoffs involved. Under Galluscio's approach, the queue is divided up between the various hosts (e.g., processes 21 and 22) so that each host can be assigned only those services it is equipped to handle. The Galluscio approach also relies on the use of "offsets" to refer to the commonly shared memory 24 so that, for example, an offset of "five" from memory segment "two" for one host or process would specify the same physical location in the shared memory as an offset of "five" from memory segment "four" for another host or process (see col. 5, lines 30-46). This is done because each host or process with its unique resources will likewise possess a different "virtual" map of the shared memory 24. However, such an approach completely defeats Cranston's vision of "process agnostic" memory handles using a shared queue where the same handle always refers to a particular physical memory location regardless of which process pulls the handle. Cranston and Galluscio, in other words, are fundamentally incompatible systems where the basic features of one cannot be applied to the other without rendering inoperable the essential functions of each (Cranston loses its process agnostic handles, bidirectional operability, and shared queues if combined with Galluscio whereas Galluscio, if combined with Cranston, loses its ability to function in distributed environments in which the hosts or processes have differing servicing capabilities and commensurate differences in their virtual memory mapping systems). In short, there would have been no apparent reason, at the time of invention, for one of ordinary skill to make the proposed combination of Cranston and Galluscio.

IVd. The subject matter of each pending claim, claims 1-45, patentably defines over the proposed combination of Cranston and Galluscio.

It has now been described how certain limitations found in subparagraph (a) of each pending independent claim (claims 1, 16, and 31) patentably defines over the proposed combination of Cranston and Galluscio and further how there would not have been any apparent and acceptable reason for one of skill to make this combination. Since the remaining claims each depend, directly or indirectly, from one of these independent claims, they too carry the same patentably defining limitations. Accordingly, all of the pending claims, claims 1-45, patentably define over Cranston in view of Galluscio and stand in condition for immediate allowance, which action is respectfully requested.

V. Even apart from the patentability of their respective base claims, dependent claims 13, 28, and 43 patentably define over the proposed combination of Cranston and Galluscio.

Claims 13, 28, and 43, target a specific application of applicant's invention where the "plural computer applications" that share the object space pertain to "at least one" of stock trading, communications processing, and internet services. The common theme here is use of applicant's claimed invention in various environments where increases in computing speed and economy of resource use are linked to substantial financial incentive (see, generally, applicant's original disclosure, from the bottom paragraph of page 15 to the end of page 16). Notwithstanding the absence of mention of these specific environments at col. 3, lines 14-36 of Cranston, doubtless at least some form of interprocess communication tool has been used to facilitate message passing in each of these contexts. Be that as it may, the real question is whether Cranston and Galluscio, even assuming it would have been obvious to rely on their message passing tools in such contexts, provide the claimed shared object space (this feature being principally responsible for the gains in speed and resource economy) as well as the claimed virtual machine operation (this feature providing the necessary infrastructure to sufficiently support the shared object space feature including, for example, through use of a method area and class loader subsystem). Cranston and Galluscio, taken either separately or together, show neither.

VI. Even apart from the patentability of their respective base claims, dependent claims 14, 29, and 44 patentably define over the proposed combination of Cranston and Galluscio.

These claims specify, in the context of an object space shared by plural applications, that at least one of the objects is copy shared among these applications. If anything, at col. 3, lines 14-36, Cranston teaches away from the use of copy sharing to move or pass objects or any other form of data. Instead Cranston teaches the use of a "process agnostic memory handle" to "generate a valid memory pointer" which can then be used by the servicing process for "direct access to [the proper location in] shared memory" (that is, the location where the pertinent instruction or operation code together with the data to be operated on is stored). Cranston observes that this eliminates the need for what it calls "data marshalling" (involving copying such data to and retrieving such data from some form of intermediate memory space; see col. 2, lines 25-33). The basic idea behind Cranston's approach is that it takes less memory to move or pass a handle or other pointer value than the multiple bytes of memory needed to move an operation identifier and associated operands. In applicant's claimed system, on the other hand, that is, in a context where the object space is being shared by plural applications, the use of a copy sharing operation to transfer objects between applications prevents inadvertent overwriting of the original objects and therefore provides another level of protection for preserving original object integrity (e.g., refer to applicant's disclosure, page 19, top paragraph). Preserving the character of the single-use instructions or executables of Cranston, in contrast, is not a concern.

VII. Even apart from the patentability of their respective base claims, dependent claims 7, 22, and 37 patentably define over the proposed combination modifying Cranston in view of Galluscio and further in view of Jaworski.

As now written (see the last paragraph of section II above), these claims require that the "shared" object space be connected to each of plural virtual machines through a Native Method Interface. Page 984 of Jaworski describes "shared" dynamic link libraries that connect with plural Java programs (and hence plural virtual machines) through a Java Native Method Interface (e.g., when each application is ported to the native operating system in the "Prior Art" manner suggested by applicant's FIG. 4A). In other words, referring to the "Prior Art" virtual machine shown in FIG. 2 of applicant's drawings, Jaworski adds nothing more than that the "native method libraries" 36 there depicted can be connected, through native interface 34, not just to one virtual machine, as explicitly shown, but to more than one virtual machine (the phrase "dynamic link" used by Jaworski in referring to this native library merely recognizes that native methods can be dynamically linked, during runtime, to reference one another, just as Java methods can be dynamically linked together).

While applicant agrees that it was conceivable, under the prior art, for plural virtual machines to "share" native method (or "dynamically linked") libraries by connection through a Native Method Interface, applicant does not agree with the underlying premise that these "shared dynamic link libraries" equate to a "shared object space." These libraries contain "native method" definitions and, to the extent these native methods are directly loaded into each virtual machine during runtime, are the native equivalent of the "Java" method definitions loaded into the method area 24 (where possible, however, the standard procedure would have been to work with the native method indirectly by invoking the corresponding method, where available, in the Java API library). Though these native or Java methods may potentially be referenced to define or extend a particular object's attributes and behavior (or, if you will, serve as a "blueprint" for such attributes and behavior), they do not represent the object itself. The runtime instance of the object is effectively created (that is, the object is "instantiated") when space is allocated for that object on the object heap 26 (FIG. 2) which, in turn, involves declaring and defining the name and data type (e.g., classtype) of the object (e.g., by invoking the "new()" or, for a directly invoked native method, the "new Native ()" constructor of the desired class). In other words, though, as Jaworski suggests, it was known in the prior art that plural virtual machines could draw upon a "shared" native method library to define one or more of their respective objects, each object was still created and stored in a *separate* object space of each virtual machine (not as, claimed, a *shared* object space connected to plural virtual machines).

It is worth mentioning that, according to conventional thinking at the time of applicant's invention, not only would it have been considered risky to provide a "shared object space" connectible to plural applications (because, as discussed in section VII above, this would have been seen as heedlessly exposing one application to runtime failure or other faulty or hostile behavior by the other application), it would have been regarded as *especially* risky to make such connection "through a Native Method Interface," as claimed. This is so because native methods directly loaded into a virtual machine through a native method interface (versus indirectly worked with by invoking a corresponding class method in the Java API library) bypass the protections conventionally afforded by the class loader subsystem 16 (FIG. 2).

Since at least before the time of applicant's invention, a class loader implementation preferably included a number of discrete class loaders each designed to load class definitions from a different library or source (see, generally, applicant's original disclosure, starting with the first paragraph under "Detailed Description"). Depending on the trustworthiness of its source, a particular class loader could then be given priority in loading class definitions; for example, if a class method was invoked, the virtual machine might first attempt to load that class through the most trusted "primordial" or "bootstrap" loader, then through the standard Java application interface (API) loader, then through the standard extensions class loader, then through any user-defined "class path" loaders, and finally through the least trusted "network" class loaders. In this manner, only the most trusted version of a class method definition was allowed to gain entry into the virtual machine (or, to express it another way, a more trusted version was prevented from being inadvertently or maliciously displaced by a less trusted version). Moreover, a separate "namespace" was maintained for each loader that listed all the classes that had been loaded by that particular loader. This made it possible to identify any class which nominally (based on name alone) seemed to belong to a more trusted group but which had been loaded by a less trusted loader (such as a "network" loader) and to block that class from gaining the special access normally accorded to members of the same group. Passing native or other methods directly through a native method interface (as opposed to working with only native-type methods that had first been vetted through a class loader of this general sort) would have negated this entire system of conventional safeguards. This is why the claimed architecture, in which each of the plural virtual machines (or applications) is connected to the shared object space *through a Native Method Interface*, would not have been obvious at the time of applicant's invention.

VIII. Even apart from the patentability of their respective base claims, dependent claims 12, 27, and 42 patentably define over the proposed combination modifying Cranston in view of Galluscio and further in view of Jaworski.

Read in the context of their respective base claim, these claims require a *shared object* space connectable to each of plural *applications each* operating *its own virtual machine*, where such shared object space is operably connectable, moreover, to a *Sun Microsystems* virtual machine. The pertinent question, then, is not whether the prior art shows a shared "instruction" or executable "message" space connectible to plural "processes" (as Cranston and Galluscio might suggest) even if such a space and one or more related queues were added as another functional block (e.g., to provide a modified form of Remote Method Invocation) to the runtime memory area of a conventional Sun Microsystems virtual machine (as Jaworski might remotely suggest; note this modified machine would still retain its conventional or "separate" object space).

IX Conclusion

It has now been shown that each of the pending claims 1-45 patentably defines over the cited art. In particular, it has been shown that each independent claim, claims 1, 16, and 31, patentably defines over Cranston and Galluscio, either taken separately or in combination. It has further been shown that certain of the dependent claims, claims 7, 12-14, 22, 27-29, 37, and 42-44, even apart from their base claims, patentably define separately over the proposed combination of Cranston and Galluscio or, where cited, Cranston, Galluscio, and the Jaworski article. Accordingly, all the pending claims stand in present condition for allowance, which action is respectfully requested.

If the examiner believes that a conference with applicant's undersigned representative would help advance this application to issue, he is invited to contact such representative at the telephone number provided below.

Applicant submits that no fees are required for entry of this Amendment. If it is determined that any fees are due, the Commissioner is hereby authorized to charge such fees to Deposit Account No. 03-1550.

Respectfully submitted

CHERNOFF, VILHAUER, MCCLUNG & STENZEL

Dated: May 4, 2007

By [illegible]

Kevin L. Russell, Reg. No. 38,292
601 SW Second Avenue, Suite 1600
Portland, OR 97204
Tel: (503) 227-5631

Certificate Of Mailing

I hereby certify that this correspondence is being deposited with the United States Postal Service as first class mail in an envelope addressed to: Mail Stop Amendment, Commissioner for Patents, P.O. Box 1450, Alexandria, VA 22313-1450, on May 4, 2007.

Dated: May 4, 2007

[illegible]
Kevin L. Russell